Carter Roberts 3/25/2025 cmrober1@my.loyno.edu Loyola University New Orleans

Operating System Midterm exam

1. For each of the following process state transitions, say whether the transition is legal and how the transition occurs or why it cannot.
   1. Change from thread state WAIT to thread state RUNNING

This transition is legal. This transition happens if a thread that had to stop for some reason, like an I/O request, is not only *able* to run again, but is also at the best time for the CPU’s scheduling to run again. To perform this transition, the CPU scheduler moves the thread’s queue to a position where it can run; typically (but not always) the topmost layer.

* 1. Change from thread state RUNNING to thread state WAIT

This transition is legal. This transition happens if a thread has to halt until some procedure is complete, like if an I/O transaction needs to complete. To perform this transition, the CPU scheduler moves the thread’s queue out of the position to run or stand ready to run, and its registers and data are temporarily stored in the kernel stack.

* 1. Change from thread state READY to thread state WAIT

This transition is legal. This transition happens if a thread that is standing by ready to run is interrupted by another thread with higher priority on running. To perform this transition, the CPU scheduler puts the thread’s queue in a position to wait, and re-stores its registers and data to the kernel stack.

1. Write a program that opens a file (with the open () system call) and then calls fork () to create a new process. Can both the child and parent access the file descriptor returned by open ()? What happens when they are writing to the file concurrently, i.e., at the same time?

To see the program, refer to the repo file “openfork.c”. Both the child and parent can access the file descriptor returned by open(), and can write to the file concurrently, but because they have equal priority in doing so, the calculation of who writes when is *nondeterministic*, in other words, it’s up to chance and unpredictable, thus not a wise idea for programming.

1. Write another program using a fork (). The child process should print “hello”; the parent process should print “goodbye”. You should try to ensure that the child process always prints first; can you do this without calling wait () in the parent?

To see the program, refer to the repo file “hibyefork.c”. It is impossible to do this without calling wait() in the parent case; wait() is the only way to counter the nondeterminism of running parent and child processes.

1. Write a program that creates a child process, and then in the child closes standard output (STDOUT FILENO). What happens if the child calls printf () to print some output after closing the descriptor?

To see the program, refer to the repo file “outputclose.c”.If the child calls printf() to print output after closing the descriptor, it will fail to print anything, because the child process was the process that closed standard output.

1. Consider the following piece of C code:

void main ( ) {

fork ( );

fork ( );

exit ( );

}

How many child processes are created upon execution of this program?

To see the program, refer to the repo file “twoforks.c”. After the first fork(), there is one child process. However, both the child and the parent process then process the second fork(), which creates two extra child processes. This means that, upon the execution of the program, there are three child processes. I used printf() to validate this result in the program.

1. An interactive shell program such as bash shell (terminal in Linux and Mac) or PowerShell or CMD prompt in Windows takes command line input from the user and then executes the command/program specified by the user. In this exercise, you will implement closh (Clone Shell), a simple shell-like program designed to run multiple copies of a program at once.  
   Like any other shell, closh takes as input the name of the program to run (e.g.,  
   hello world). However, closh also takes two additional inputs:
2. The number of copies (processes) of the program to run. This is an  
   integer from 1 to 9.
3. Whether the processes should be executed concurrently or  
   sequentially, in sequential execution, the shell should wait for every  
   time a program is executed. In case concurrent execution, the shell  
   does not need to wait for the program to complete execution.

Closh executes the given program, the specified number of times, then  
returns to the prompt once all processes have either completed. Here is a  
simple example of using closh

(italic is user input and hello.exe is the hello world as in previous question):  
mint@mint:~$ ./closh  
closh> ./hello.exe  
count> 3  
[p]arallel or [s]equential> p  
hello world  
hello world  
hello world  
closh>

To write such a shell in C++, refer to the process lecture. As noted in the slide, you can use “execvp()” to create a new child process and have it execute a command. Use also the “waitpid” in the parent process to wait for the child process to finish in case  
sequential execution is selected.  
For simplicity, assume that the user specifies the full path name for any  
command/executable that they wish to execute. Thus, you do not need to deal  
with path name completion issues. You can test your shell on programs you  
write in C++ (after compiling to machine language).

To see the program, refer to the repo file “roberts\_closh.cpp”. Thank you!

Create a GitHub repository and upload this document with answers to question 1 and 6 with all your .cpp or .c programs. Copy the repository link and paste it in Canvas assignment.